Dealing with it - an easier way to play with cards

Jemima Lomax-Sawyers, Reuben Sweetman-Gough, George Zotta

The count required to return the pile to its original state when picking up by rows is constant - for the LT specification, it is only 1, which one can see intuitively given the pile is picked up and laid down in exactly the same way. For the other row wise transformations, the count is always 2. The effect of the length of the row or the number of columns does not impact this. On the other hand, picking up in a column wise fashion will cause counts of varying values, and will be influenced by the row length and specification. A fairly obvious reason for this is the priority with which the cards are laid down and picked up - it being the same. Indeed, if the priority by which the cards were laid down were in a column wise fashion, one would observe the opposite effect. The pile behaves as a queue of first in, first out - the first card to be picked up is the first to be placed down. Furthermore, the two dimensional data structure in which the pile is laid down (and subsequently transformed) can be thought of as a linked list, with both horizontal and vertical pointers. The behaviour of the algorithm does not change when repeated, which is the key point. And when implementing the transform method in a row wise specification, the pointers of a card do not change - that is, while pointers may swap, the neighbours themselves always remain the same. Hence, if the algorithm is repeated, the pointers will always swap back into their original positions, generating a count of exactly 2 (with the exception of the LT specification, as no pointers are swapped).

The maximum count value produced for any specification and any pile size of 20 or fewer is 18. The answer can be produced with the pile size, row length, and specifications of a pile size of 20, with either a row length and specification of 2 and TL or BR, 10 and TL or BR, 5 and BR, or 4 and BR. Alternatively with a pile size of 18, with either a row length and specification of 2 and TR, 9 and BL, 6 and BL, or 3 and TR.

Finding the count without carrying out every transformation can be approached by considering the algorithm as a type of shuffle that rotates in a circular manner. The basis for this approach is an implementation of cyclic permutations. Essentially, after a single transformation for any particular pile size, row length, and specification, it can be seen that every element in the pile is mapped to an element from the original array. This is not necessarily an indication that these two elements will swap positions each time (although this does happen in certain circumstances). Rather, it means that with every transformation, a card will change to the position of the card it is mapped to in the previous array. These circular permutations will continue until the point at which all cycles are complete - that is, each element has resumed its original position. Putting

this approach into practice, one need only complete a single transformation, and then apply the principles behind cyclic permutations to implement a much faster algorithm that determines the minimum number of transformations based on this circularity. The algorithm itself relates to the connected components of the array, and the count of nodes that are reachable by a node, finding the least common multiple (for which one needs to determine the greatest common divisor (an algorithm that can be implemented recursively). With this approach, the only necessary transformation is the first.

There are several brute force approaches one might take to determine the number of accessible for a pile of size n. None of these would be feasible in that while one may reach the correct answer at some point, there is no way to check the validity of the answer. It leads to the problem of the inability to know when to terminate the algorithm - that is, when to stop trying to find new piles.

Some approaches are better than others in terms of efficiency, however all require knowledge of a particular pattern of behaviour of the pile. One example would be to check against every possible permutation of the pile, however this would quite possibly be the least efficient way to do this, as it would have a time complexity of O(n!), causing the scale at which the number of required executions grows to be extremely rapid, and quickly becoming impractical for even relatively small values of n.

The other approach might be to perform sequences of transformations, and each time a new one is generated, to add it to a list of all of the piles so far. A pile will only be added to the list once it has been checked against every pile currently in the list to determine whether it is distinct. Again, this would require knowledge of when to terminate the algorithm.

A hint to what factors play into the number of accessible piles of any given pile size relates to piles that have a length that is a prime number. That factor being, fittingly, the number of factors of n. A prime number has exactly two factors, and has a constant number of accessible piles of 2. This is not to say that a pile will always have this number being its number of factors, as this is not the only determinant, but it does lead us to theorise that the factors of the pile are relevant.

The factors of a number can be determined when looping through the numbers of 1 to n with a time complexity of O(n). This can, however, be optimised further. We only really need the number of pairs of factors - as using both factors will not generate a larger number of distinct piles than will using only one of the two. An algorithm finding distinct factors of n only has a time complexity of O(sqrt(n)).

In theory, to generate unique piles, one need only use three types of transformations, as each pile has a corresponding specification of which it is the inverse. We can also rule out one of the pairs of transformations, LT and RB, as they do not fundamentally alter the structure of the pile. Looping through the transformations for a specification to return to the original pile at worst requires n iterations. We theorise that to generate piles to consider, the upper bound of the number of required iterations is n^3 - based on there being 3 unique transformations to consider. The number of factors will also play into this, however should not be important when determining the time complexity, as the “worst” part of the algorithm is the n^3 component.

Another consideration for the problem is the observation referenced previously of each card being mapped to another card, and the behaviour of the transformations being in cyclical fashion. In theory, if one might calculate the number of nodes any one card can possibly reach, the number of possible permutations (accessible piles, if you will) can be determined. This is, however, more complex than in the previous discussion relating to its use in generating counts, as the number of reachable nodes is far greater, and more time intensive to determine - for every combination of transformation, the mappings of a node will change, leading to a very different number of reachable components.

In summary, we believe it would be difficult to greatly optimise the time complexity of an algorithm that determines accessible piles, and that it would be insufficient to deal with larger numbers - in fact, the time complexity, even if not n!, is still such that it would quickly become slower as the pile size increased, and would be impractical to use even for moderately larger numbers. An effective solution to the problem also requires an understanding of the behaviour of the pile, several theories of which we have alluded to.